ІНФОРМАТИКА ТА ПРОГРАМУВАННЯ

Тема 16. Ітератори та генератори

Ітератори

- Ми вже неодноразово зустрічались з об’єктами складених типів даних, які допускають перебір всіх елементів за допомогою циклу for … in … .
- Ці об’єкти ми називали такими, що ітеруються.
- Раніше це були об’єкти стандартних типів Python.
- Але є можливість будувати і свої власні об’єкти та навіть надавати такої поведінки (здатності до перебору всіх елементів) раніше побудованим типам.
- Для цього використовують ітератори.
- Нехай *x* – об’єкт типу, що ітерується (це, зокрема, означає, що *x* складається з декількох елементів).
- Тоді **ітератором** *y* називається об’єкт, який здатен повертати по черзі всі елементи *x* у деякому порядку та фіксувати момент завершення елементів *x*.
- У такому випадку кажуть, що ітератор *y* підтримує ітераційний протокол.

Функції next() та iter()

- **Ітераційний протокол** полягає у наступному.
  - Для повернення елементів *x* ітератор *y* використовує стандартну функцію next(y).
  - Коли елементи *x* завершуються, ітератор у відповідь на черговий виклик next ініціює виключення класу StopIteration.
- Для того, щоб для *x* повернути об’єкт-ітератор, використовують стандартну функцію iter(x).
- Після цього послідовно викликаючи next, можна отримати всі елементи *x*.

Функції next() та iter().2

- Цикл for … in … також використовує ітераційний протокол.
- Цей цикл рівносильний такій інструкції:

```
for a in x:     ≡      y = iter(x)
    P                  try:
                           while True
                               a = next(y)
                               P
                       except StopIteration:
                           pass
```

Власні класи-ітератори

- Для реалізації власних ітераторів треба описати клас, який реалізує методи __iter__ та __next__.
- При виклику функції iter(x) Python буде викликати метод __iter__: x. __iter__().
- Як правило, цей метод повертає у якості ітератора самого себе, тобто, об’єкт цього класу.
- При виклику функції next(x) Python викликає метод __next__: x.__next__().
- Метод __next__ повинен повертати елементи та ініціювати виключення StopIteration, якщо елементи завершились.

Приклад

- Клас-ітератор Reverse, що повертає елементи послідовності в оберненому порядку.

Генератори

- **Генератори** – це об’єкти, які створюють послідовність поелементно та повертають по одному елементу послідовності за один крок.
- Генератори схожі на ітератори.
- Але якщо ітератор повертає елементи вже існуючої послідовності, то генератор цю послідовність створює.
- Генератори у Python синтаксично можуть бути реалізовані як **генератори-вирази** або як г**енератори-функції**.

Генератори-вирази

- Синтаксис генератора-виразу виглядає так:

(*e*(a) for a in *x* if *F*)

  - де *e*(a) – вираз, що залежить від a, *x* – вираз типу, що ітерується, *F* – умова.
- Python повертає значення для всіх елементів a з *x*, для яких істинною є умова *F*.
- Умову *F* можна не вказувати. Тоді if *F* опускають.
- Генератор-вираз схожий на спискоутворення.
- Але окрім різних дужок у синтаксисі суттєвою відмінністю є те, що генератор-вираз не будує всю послідовність, а повертає її поелементно.

Генератори-функції

- Генератори-функції мають такий же синтаксис, як і звичайні функції, за виключенням того, що для повернення результату замість оператора

return e

використовують

yield e

- З точки зору виконання функції yield відрізняється від return тим, що не тільки повертає значення виразу e, але й запам'ятовує стан функції (місце завершення, значення всіх локальних змінних).
- При наступному виклику генератора-функції її виконання починається з наступного оператора після yield.
- Тобто, при багатьох викликах генератора-функції управління виконанням програми перемикається від програми до генератора-функції і назад, аналогічно тому, як це відбувається при паралельному виконанні програм.
- Тому генератори-функції ще називають «паралельним програмуванням для бідних».

Приклад

- Отримати всі числа Фібоначчі в діапазоні від 1 до n. Використати генератор-функцію.

Застосування генераторів

- Генератори не дають якихось унікальних переваг у порівнянні з іншими засобами роботи з послідовностями, окрім випадків, коли треба обробляти великі обсяги даних і коли, можливо, з великого масиву даних достатньо отримати кілька елементів.
- Тому їх застосування є доцільним саме при наявності послідовностей великого розміру.
- Варто зазначити, що результати раніше розглянутих функцій zip() та map() є саме об’єктами генераторами

Приклад

- Побудувати всі перестановки з n заданих елементів.

Реалізація ітераторів у класах

- Один із способів побудови власних ітераторів – це додавання підтримки ітераційного протоколу у існуючих класах або їх нащадках.
- Для цього необхідно реалізувати у класі методи
  - __iter__ та __next__
  - або навіть один метод __iter__, який повертає об’єкт, що підтримує ітераційний протокол.
- Наприклад, генератор-функцію.

Приклад. Ітератор для графу

- Побудувати ітератор для графу та перевірити, чи є граф деревом.
- У темі «Рекурсивні структури даних» ми розглядали реалізацію орієнтованих графів на базі списку (клас Graph).
- Опишемо клас-нащадок цього класу – GraphIt, який реалізує ітератор, що проходить по всіх вершинах графу.
- Метод __iter__ є генератором-функцією, яка підтримує ітерайційний протокол. Тому метод __next__ реалізовувати непотрібно.
- Клас GraphIt також містить методи обчислення напівстепіні входу та виходу заданої вершини.
- Ці методи використовуються для перевірки того, чи є граф деревом.
- За означенням, граф є деревом, якщо він є зв’язним, має одне джерело та напівстепінь входу кожної вершини не більше 1.

Приклад. Ітератор для графу.2

```
class GraphIt(Graph):
    '''Реалізує граф з ітератором по вершинах.

    '''
    def __iter__(self):
        '''Повернути елемент.
        '''
        for k in self.nodes():
            yield k

    def hdegin(self, key):
        '''Напівстепінь входу вершини.
        '''
        return len(self.getpredecessors(key))

    def hdegout(self, key):
        '''Напівстепінь виходу вершини.
        '''
        return len(self.getsucceders(key))
```

Написання власних класів-ітераторів

- Можна описати власні класи-ітератори для типів даних, які складаються з елементів, але не є такими, що ітеруються.
- Для цього потрібно описати клас, у якому реалізувати методи __iter__ та __next__.
- Метод __iter__ у цьому випадку повинен повертати себе в якості об’єкта-ітератора, а метод __next__, - забезпечувати перебір всіх елементів та ініціювання виключення StopIteration.

Обхід бінарного дерева

- Як приклад наведемо бінарне дерево, опис та реалізацію якого ми розглядали у темі «Рекурсивні структури даних».
- Бінарне дерево не належить до типів, що ітерується, але можна реалізувати ітератор, що буде повертати всі вершини дерева у певному порядку.
- Перебір всіх вершин дерева у заданому порядку називають обходом дерева. Існують різні типи обходів, серед яких виділяють 3: КЛП, ЛКП та ЛПК.
- Обхід КЛП дерева t (скорочення від корінь-лівий-правий) визначає порядок обходу дерева, при якому спочатку проходять корінь t, потім всі вершини дерева, що є лівим сином t, а потім, - всі вершини дерева, що є правим сином t.

Обхід бінарного дерева.2

- Обхід ЛКП дерева t (скорочення від лівий- корінь- правий) визначає порядок обходу дерева, при якому спочатку проходять всі вершини дерева, що є лівим сином t, потім корінь t, а потім, - всі вершини дерева, що є правим сином t.
- Обхід ЛПК дерева t (скорочення від лівий -правий- корінь) визначає порядок обходу дерева, при якому спочатку проходять всі вершини дерева, що є лівим сином t, потім всі вершини дерева, що є правим сином t, а потім, - корінь t.
- Означення обходу очевидно є рекурсивним, оскільки визначений порядок обходу рекурсивно застосовується для дерев, які є лівим та правим сином t.

Реалізація класів-ітераторів для бінарного дерева

- Опишемо три класи-ітератори для здійснення обходу бінарного дерева: KLP, LKP, LPK.
- Використаємо стек (поле _st), також описаний у темі «Рекурсивні структури даних», для збереження всіх піддерев бінарного дерева таким чином, щоб дерево, корінь якого ми проходимо першим, лежав у верхівці стеку.
- Тоді метод __iter__ просто повертає себе, а метод __next__ бере та повертає дерево, що лежить у верхівці стеку, зупиняючись, коли стек стає порожнім.
- Сам стек наповнює внутрішній метод класу _populate().
- При такому підході виявляється що класи KLP, LKP, LPK відрізняються тільки реалізацією метода _populate(), а методи __iter__ та __next__ у них спільні.
- Тому класи LKP та LPK можуть бути нащадками класу KLP.

Реалізація класів-ітераторів для бінарного дерева.2

```
class KLP:
    """Ітератор для реалізації обходу КЛП.
"""

    def _populate(self, t):
        """Наповнити стек піддеревами дерева t у порядку КЛП.
        """
        if not t.isempty():
            self._populate(t.rightson())    #спочатку правий син
            self._populate(t.leftson())     #потім лівий син
            self._st.push(t)                #потім корінь


    def __init__(self, t):
        self._st = stack.Stack()    #_st - стек, що містить піддерева дерева t
        self._populate(t)
```

Реалізація класів-ітераторів для бінарного дерева.3

```
    def __iter__(self):
        return self

    def __next__(self):
        if self._st.isempty():      #якщо стек порожній
            raise StopIteration     #зупиняємось
        else:
            t = self._st.pop()      #інакше повертаємо піддерево з верхівки стеку
            return t
```

Реалізація класів-ітераторів для бінарного дерева.4

```
class LKP(KLP):
    """Ітератор для реалізації обходу ЛКП.

    """
    def _populate(self, t):
        """Наповнити стек піддеревами дерева t у порядку ЛКП.
        """
        if not t.isempty():
            self._populate(t.rightson())    #спочатку правий син
            self._st.push(t)                #потім корінь
            self._populate(t.leftson())     #потім лівий син
```

Реалізація класів-ітераторів для бінарного дерева.5

```
class LPK(KLP):
    """Ітератор для реалізації обходу ЛПК.

    """
    def _populate(self, t):
        """Наповнити стек піддеревами дерева t у порядку ЛПК.
        """
        if not t.isempty():
            self._st.push(t)                #спочатку корінь
            self._populate(t.rightson())    #потім правий син
            self._populate(t.leftson())     #потім лівий син
```

Приклад

- Показати список вершин дерева для кожного з обходів КЛП, ЛКП, ЛПК.
- Для реалізації показу списку вершин використаємо описані у прикладі для бінарного дерев у темі «рекурсивні структури даних» функції makewords – отримати список слів з перемішуванням, та build tree – побудувати дерево пошуку.

Обхід бінарного дерева за допомогою генераторів-функцій

- Іншим способом реалізації ітераторів для обходу бінарного дерева є застосування генераторів-функцій.
- Відмінність цих функцій від раніше розглянутих генераторів полягає у тому, що вони повинні бути рекурсивними.
- Скажімо, для обходу КЛП, треба після повернення кореня рекурсивно викликати той же генератор для повернення всіх вузлів лівого, а потім – правого сина.
- Для рекурсивного виклику генератору-функції f у Python використовують
- `yield from f()`

Реалізація генератору-функції для обходу КЛП

- Реалізація генератору-функції для обходу КЛП – нижче.
- Генератори-функції для ЛКП та ЛПК реалізуються аналогічно.

Реалізація генератору-функції для обходу КЛП.2

```
def klp_gen(t):
    '''Генератор-функція, що повертає всі вузли t у порядку КЛП
    '''
    if not t.isempty():
        yield t.root()                    #повернути корінь дерева
        yield from klp_gen(t.leftson())   #рекурсивний виклик генератора для лівого сина
        yield from klp_gen(t.rightson())  #рекурсивний виклик генератора для правого сина
```

Резюме

- Ми розглянули:
  1. Ітератори, функції iter та next
  2. Ітераційний протокол
  3. Генератори
  4. Генератори-вирази та генератори функції
  5. Написання власних класів-ітераторів

Де прочитати

1. Марк Лутц, Изучаем Python, 4-е издание, 2010, Символ-Плюс
2. Python 3.4.3 documentation
3. Марк Саммерфилд, Программирование на Python 3. Подробное руководство. - Символ-Плюс, 2009.
4. Bruno R. Preiss, Data Structures and Algorithms with Object-Oriented Design Patterns in Python, 2003, http://www.brpreiss.com/books/opus7/
5. Tarek Ziadé. Expert Python Programming. - Packt Publishing, 2008.
6. David Beazley and Brian K. Jones, Python Cookbook. - O'Reilly Media, 2013.
7. http://www.python-course.eu/python3_generators.php